APPLICATION FOR UNITED STATES LETTERS PATENT

ARRANGEMENTS AND METHODS FOR INVOKING AN UPCALL IN A COMPUTER SYSTEM

Inventor(s): Vishwas P. Durai
825 Kiely Blvd #22
Santa Clara, CA 95051

Entity: Large

ARRANGEMENTS AND METHODS FOR INVOKING AN UPCALL IN A COMPUTER SYSTEM

BACKGROUND OF THE INVENTION

[0001] The execution of a multi-threaded application program in a computer system generally involves a large number of threads, both in the user space and in the kernel space of the operating system. Generally speaking, the user-space threads are serviced by corresponding kernel threads in the kernel space of the operating system.

[0002] In the 1X1 thread model, there is a corresponding kernel thread for each existing user-space thread. To facilitate discussion, Fig. 1 shows a 1X1 thread model wherein an application program 102 generates three user-space threads 104, 106, and 108. These user-space threads are serviced by corresponding kernel threads 110, 112, and 114 in the kernel space of the operating system. The one-to-one correspondence between user-space threads and kernel threads are shown by the three lines 120, 122, and 124. The three kernel threads are scheduled for execution by a kernel scheduler 130 on two processors 132 and 134 of Fig. 1.

[0003] A complex application program may create hundreds or thousands of user-space threads during its execution lifetime. If a kernel thread is required for each existing user-space thread, thousands of kernel threads may exist concurrently, all of which require a large amount of resources in the kernel. Given the fact that kernel resources are expensive, the 1X1 thread model is therefore not necessarily the most efficient.

[0004] Fig. 2 shows a MxN thread model in which a smaller number (N) of kernel threads exist for a larger number (M) of user-space threads. With reference to Fig. 2, three user-space threads 202, 204, and 206 from an application program 208 are serviced by two kernel threads 210 and 212 via a user-space scheduler 214, which may implement any suitable scheduling algorithm (e.g., round-robin, weighted round-robin, modified round-robin, etc.). Another user-space thread 216 is shown having a dedicated kernel thread 218 to illustrate that the MxN thread model may also have dedicated one-to-one relationships between user-space threads and kernel threads in order to provide backward compatibility and/or to improve performance for certain user-space threads/applications.

[0005] Since fewer kernel threads are multiplexed among a larger number of user-space threads in Fig. 2, less kernel resource is required to service the application. Since kernel resources are expensive, the MxN thread model has become increasingly popular.

[0006] The MxN thread model however requires a facility in the kernel to invoke a routine in the user space during system call returns. This facility, called "upcall," is substantially similar to a signal handler in its requirement for the execution of a user-space routine from the kernel. Currently, the invocation of the user-space routine from the kernel is required during a system call return or a trap return from the kernel.

[0007] To facilitate discussion, Figs. 3 and 4 illustrates how a system call may be made and how a signal is handled in the system call return path in a MxN thread model system, such as in a HP-UX™ system by the Hewlett-Packard Company of Palo Alto, CA. Currently, the mechanism illustrated in Figs. 3 and 4 represents the primary mechanism available to execute a user space code segment from the kernel (i.e., by launching a signal handler).

[0008] Fig. 3 illustrates a simplified method of handling systems calls between the user-space and the kernel space. In the example of Fig. 3, a read system call is employed as an example although other system calls may work analogously. In block 302, the application makes a system call read( ), which begins as a user-space thread. The user-space thread read ( ) then jumps to the user-space stub read_sys ( ) to handle the system call in block 304.

[0009] From the user-space stub read_sys ( ), a transfer through a gateway facilitates a privileged transfer to the kernel space, which has a higher privilege than the user space. Specifically, the privileged transfer invokes a system call initialization routine syscallinit (block 306), which saves user space context of the application (i.e the contents of the register state), into a save-state.

[0010] In block 308, the high-level system call sys_call ( ) is invoked in order to, for example, marshal the necessary arguments for the actual kernel system call READ, which is shown in block 310. The kernel system call READ then performs the action required by the application, as specified originally in block 302. In the case of READ, the reading of data may involve waiting until the I/O device (e.g., the hard disk or a data on a network) becomes available. In such a case, the kernel thread is "blocked" and the system call is essentially in a wait state, waiting for unblocking to occur before the system call can be completed. Reference number 312 as shown illustrates the kernel system call block in Fig. 3.

[0011] Suppose the I/O device permits reading at some point in time. Once the kernel system call READ acquires the required data, the kernel thread is unblocked and can begin to initiate the system call return process. In block 314, the context data, which is previously saved in block 306, is restored via the syscallrtn routine. Once the registers are restored, the user-space privilege is restored to allow the thread to branch back to the user-space stub read_sys ( ) of block 304. From user-space stub read_sys ( ) of block 304, a branch is made to the original user-space system call read ( ) in block 302, which then completes the system call by the application. This is how system calls work on HP-UX™.

[0012] Generically speaking, in a MxN thread system, signal handlers are launched from the kernel by saving the current execution context onto the thread's user stack. The signal handler return frame is then pushed onto the user stack to allow the signal handler to return back to the kernel once it finishes execution of the syscall or trap return. A branch to the user-space signal handler is made, followed by execution of the user-space signal handler code. On return from the signal handler, the control frame transfers the fabricated frame, which is basically set up to execute the system call sigcleanup, which transfer control back to kernel.

[0013] Fig. 4 is a flowchart illustrating how signal handling is performed in a HPUX™ system that implements MxN threading during the system return call path. In block 402 represents a thread running in user space. At some point this thread needs to make a system call and thus enters the kernel. The thread, on entry into the kernel will save its context data in block 404 (as shown earlier in Fig.3, block 306). After completing the system call, on the return path In block 406, the kernel thread sees the signal and has to handle it. It first saves the current execution context in the kernel (block 408), including for example the current kernel registers, in order to service signal.

[0014] In block 410, the user context data stored earlier in block 404 is copied onto the user stack in the user space for restoration at a later time. In block 412, the user stack in the user space is modified to allow a system call sigcleanup frame to be pushed onto the user stack. This frame is setup to allow the clean up to take place after the signal handler finishes. Subsequently, block 414 branches to the user space. In block 416, a signal handler is launched in the user space to execute the signal handler user space code. In block 418, the sigcleanup frame, which was pushed onto the user stack earlier in block 412, is popped to enable a privileged transfer back into the kernel to restore previously saved context data in the kernel.

Thus in block 420, in the kernel space, the user context data is saved. This context data saving step is analogous to that performed in block 404 earlier.

[0015] In block 422, the execution context stored earlier in block 408 is restored. Block 422, takes the thread back to user space from where it entered the kernel. The operation described above, although for a system call, is applicable to the trap path as well.

[0016] While the approach above works fine for signal handlers, significant performance degradation is experienced if the signal handler approach is employed to handle "upcalls." To illustrate, Fig. 5 shows the steps implementing an upcall as a signal handler in a MxN thread system. In block 502, the application makes a system call that requires the use of a kernel thread. The user-space context data is saved once upon entering the kernel, as described before in block 404 of Fig. 4. In block 504, the kernel thread is blocked. With respect to the READ system call example, the kernel thread may be blocked while waiting for the I/O device to become available, for example. After the kernel thread is unblocked, the unblock handler launch setup occurs in block 506. In this case, the unblock handler is set up as a signal handler.

[0017] In block 506, the current execution context is saved, and the user-space context data stored earlier upon entering the kernel space is copied onto the user stack in the user space. The steps associated with block 506 is analogous to those described in blocks 408, 410, and 412 of Fig. 4. After block 506, a branch into the user space occurs.

[0018] In block 508, the unblock handler is launched, which results in for example the notification to the user space scheduler that unblocking of the kernel thread has occurred. The launching and execution of the unblock handler is analogous to the steps described in connection with blocks 416 and 418 of Fig. 4. After block 508, a branch back into the kernel space occurs (block 510). Also in block 510, the user context data is saved upon entering the kernel space.

[0019] In block 512, the execution context stored earlier in block 506 is restored, enabling a return to the user space (block 516).

[0020] As can be seen in Fig. 5, implementing an upcall as a signal handler requires, in addition to the saving of the user-space context upon entering the kernel space (e.g., between blocks 502 and 504 of Fig. 5), two additional save operations and a copy operation as overhead. The first save operation involves saving the current execution context in block 506.

The second save operation involves saving the user-space context upon reentering the kernel space (block 510). The copy operation involves copying the user-space context data into the user stack (block 506). These overhead operations are handled using kernel resources, which tend to be the limiting factor in most systems since kernel resources are comparatively expensive. Additionally, an application may involve hundreds or thousands of system calls in its execution lifetime. With the extra overhead on the kernel resources impacting each system call blocking and unblocking, the arrangement of Fig. 5 results in a significant degradation in system performance.

SUMMARY OF INVENTION

[0021] The invention relates, in one embodiment, to a computer-implemented method for executing an application system call, the application system call involving invoking a kernel thread from a system call stub in a user space of an operating system of the computer. There is included saving user space context data in a save state upon entering a kernel space of the operating system from the user space. There is further included modifying a return pointer in the save state to an address of a unblock handler call stub in user space instead of an address of the system call stub in the user space, thereby causing the kernel thread to return to the unblock call handler stub instead of returning to the system call stub when the kernel thread completes execution.

[0022] In another embodiment, the invention relates to an article of manufacture comprising a program storage medium having computer readable code embodied therein. The computer readable code is configured for executing an application system call. The application system call involves invoking a kernel thread from a system call stub in a user space of an operating system of the computer. There is included code for saving user space context data in a save state upon entering a kernel space of the operating system from the user space. There is also included code for modifying, after the saving, a return pointer in the save state to an address of a unblock handler call stub in user space instead of an address of the system call stub in the user space, thereby causing the kernel thread to return to the unblock call handler stub instead of returning to the system call stub when the kernel thread completes execution.

[0023] In yet another embodiment, the invention relates to a computer-implemented method, in a computer implementing MxN threading, for executing an application system call

from a user space of an operating system of the computer. The application system call involves invoking a kernel thread in a kernel space of the operating system from a system call stub in a user space of the operating system. There is included saving user space context data in a save state upon entering the kernel space. There is further included modifying a return pointer in the save state to an address of a unblock handler call stub in user space instead of an address of the system call stub in the user space, thereby causing the kernel thread to return to the unblock call handler stub instead of returning to the system call stub when the kernel thread completes execution. Additionally, there is included returning from the unblock call handler stub in the user space to the system call stub in the user space without entering the kernel space again.

[0024] These and other features of the present invention will be described in more detail below in the detailed description of the invention and in conjunction with the following figures.

BRIEF DESCRIPTION OF THE DRAWINGS

[0025] The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

[0026] Fig. 1 shows a 1X1 thread model wherein an application program generates three user-space threads.

[0027] Fig. 2 shows an MxN thread model in which a smaller number (N) of kernel threads exist for a larger number (M) of user-space threads.

[0028] Fig. 3 illustrates a simplified method of handling systems calls between the user-space and the kernel space.

[0029] Fig. 4 is a flowchart illustrating how signal handling is performed in a HPUX™ system that implements MxN threading during the system return call path.

[0030] Fig. 5 shows the steps implementing an upcall as a signal handler in a MxN thread system.

[0031] Fig. 6 illustrates, in accordance with one embodiment, the steps involved in invoking an upcall.

[0032] Fig. 7 illustrates, in accordance with another embodiment of the present invention, the steps involved in invoking an upcall.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

[0033] The present invention will now be described in detail with reference to a few preferred embodiments thereof as illustrated in the accompanying drawings. In the following description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without some or all of these specific details. In other instances, well known process steps and/or structures have not been described in detail in order to not unnecessarily obscure the present invention.

[0034] Since upcalls are invoked frequently during the execution lifetime of a typical application, especially an application that requires lots of I/O accesses, it is imperative that upcalls be made as efficient as possible. In accordance with one embodiment of the present invention, there is provided an inventive arrangement and method for invoking an upcall during a system call return in a manner so as to avoid the performance penalties associated with implementing upcall as a signal handler.

[0035] The invention, in one embodiment, circumvents the need to copy the context data onto the user stack, setting up a user stack frame, and returning to the kernel to complete system call return. In one embodiment, the upcall stub acts as a springboard to branch to the upcall handler. The upcall handler stub may be implemented in architecture-specific code, which avails architecture-specific features to further enhance performance. Additionally, the invention requires substantially no modification to the syscall stub(s), and hence raises no compatibility issues with existing systems.

[0036] In one embodiment, the launch of an upcall handler involves the installation of the upcall handler, which involves an initial setup during application startup. During the initial setup, the address of the user space upcall handler stub and the data pointer corresponding to this address is registered with the kernel. This may be done, for example, during the thread library startup, which is executed before the start of the application code.

[0037] Fig. 6 illustrates, in accordance with one embodiment, the steps involved in invoking an upcall. When the thread executes a blocking system call, the thread goes to a wait

mode or to sleep in the kernel. Before it goes to a wait mode, however, the application save-state (the register context of the thread in the user space) is modified such that the return pointer (e.g. gr31 in PA-RISC™ architecture) returns to the upcall handler stub. Further, the data pointer is modified to the upcall handler's stub's data pointer. The original data pointer and return pointer are, however, setup to be passed as arguments to the upcall handler stub.

[0038] When the thread wakes up from its wait mode or its sleep and returns from the system call, it returns to the upcall handler stub, since the save state (and more specifically the return pointer in the save state) is modified to return to the upcall handler stub. The upcall handler stub then acts as a springboard to jump to the real upcall handler. The real upcall handler is a notification mechanism to the user space scheduler. After the real upcall handler is executed, the thread should prepare to branch back to the system call stub, from where it started. So the system call return values, the system calls stub's data pointer, and the return pointer of the system call stub are restored. A branch to the return pointer of the syscall stub then transfers control to the syscall stub. At that point, the syscall stub will continue as if a normal return from the system call has occurred.

[0039] Fig. 7 illustrates, in accordance with another embodiment of the present invention, the steps involved in invoking an upcall. In Fig. 7, the line 700 separates the user space (above line 700) from the kernel space (below line 700). In block 702, the application makes a system call (e.g., read). As before, the user space thread jumps to a user space system call stub (block 704) to facilitate a transfer to the higher privilege kernel space. During the transfer process to the higher privilege kernel space, the user space context data is saved as discussed earlier.

[0040] In block 706, the kernel thread executing the system call (e.g., read) is invoked. As mentioned, this kernel thread may be blocked due to, for example, the unavailability of the I/O device (e.g., the hard disk). Accordingly, the thread is essentially in a sleep or wait mode, waiting for unblocking to occur. At some point, unblocking occurs (e.g., the I/O device becomes available). At that point, the return pointer is set in block 708 to the address of the unblock handler stub. This modification is possible since the kernel knows in advance (e.g., during registration) the address of the unblock handler stub. The original return pointer (i.e., the pointer pointing to the syscall stub) is passed as argument to the user space unblock handler stub. Further, the data pointer is modified to the upcall handler's stub's data pointer. and the

original data pointer, the system call return value and the original system call stub address is passed as an argument to the user space unblock handler stub.

[0041] In block 710, the thread branches to the address of the newly modified return pointer and returns to the unblock handler stub (block 712) instead of the syscall return stub. This is possible because, as mentioned earlier, the return pointer has been modified to point to the unblock handler stub instead of to the syscall return stub. The branch to the unblock handler stub accomplishes the transfer to the lower privilege user space. At this point, the unblock handler stub acts as a springboard to transfer the thread to the unblock upcall handler. In the MxN thread scenario, the unblock upcall handler may involve, for example, notifying the user space scheduler.

[0042] In block 716, after the user space unblock upcall handler is completed, a branch to the original return pointer, i.e., the return pointer associated with the user space syscall stub transfers control to the syscall stub (path 718). This is possible because the syscall return values, the syscall stub's data pointer, and the return pointer of the syscall stub, were passed to the unblock handler stub as arguments earlier.

[0043] After the user space unblock upcall handler is completed, these values are restored. After branching to the syscall stub, the syscall stub at that point behaves as if it has just experienced a return from a kernel system call, which may include returning to the original system call (via path 720) to conclude the system call.

[0044] Note that unlike the implementation of the upcall as a signal handler, there is no need to re-enter the kernel space after the user space upcall handler completes execution in order to complete the system call. This eliminates two context save operations and one context data copy operation. In comparison with Fig. 5, for example, the saving of the current execution context (in block 506) is no longer necessary. The second save operation involves saving the user-space context upon reentering the kernel space (block 510) is also not necessary. Further, the copy operation involves copying the user-space context data into the user stack (block 506) is also eliminated. Accordingly, expensive kernel resources are not wasted on these now-eliminated operations and performance is vastly improved, which further the goal and philosophy behind the MxN thread architecture.

[0045] While this invention has been described in terms of several preferred embodiments, there are alterations, permutations, and equivalents which fall within the scope

of this invention. For example, although the invention works well with systems implementing the PA-RISC™ and IPF™ architectures (available from the aforementioned Hewlett-Packard Company and Intel Corporation of Santa Clara, CA respectively), it is contemplated that the invention may work with any suitable systems implementing MxN threading. It is therefore intended that the following appended claims be interpreted as including all such alterations, permutations, and equivalents as fall within the true spirit and scope of the present invention.